CS152 Computer Architecture and Engineering

Bus-Based RISC-V Implementation

General Overview

Figure H1-A shows a diagram of a bus-based implementation of the RISC-V architecture.[1] In this microarchitecture, the different components of the machine share a common 32-bit bus through which they communicate. Control signals determine how each of these components is used and which components get to use the bus during a given clock cycle. These components and their corresponding control signals are described below.

For this handout, we shall use a positive logic (active-high) convention. Thus, when we say signal X is "asserted", we mean that signal X is a logical 1, and that the wire carrying signal X is raised to the "HIGH" voltage level.

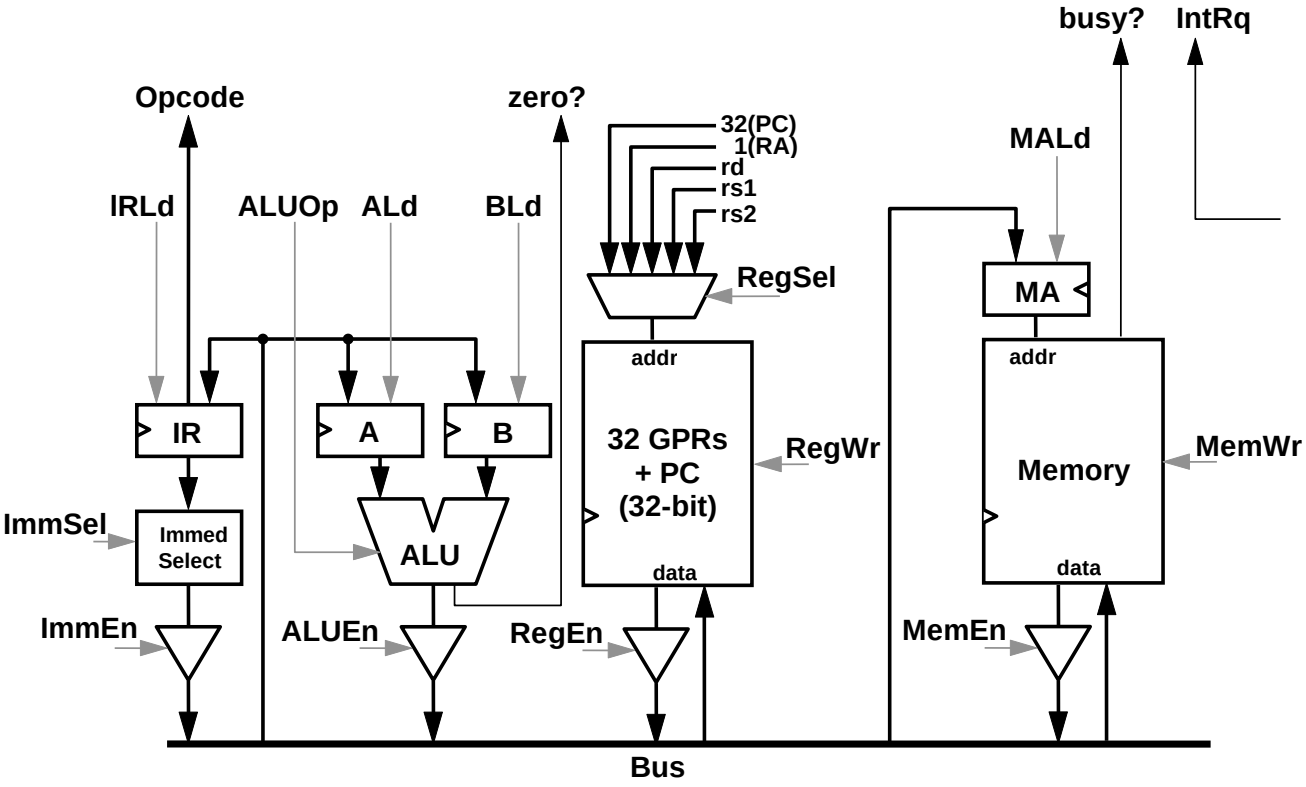


Figure H1-A: A single-bus datapath for RISC-V.

[1] This document discusses an implementation of RV32, the 32-bit address space variant of RISC-V. RV32 uses 32-bit wide general-purpose registers and a 32-bit wide PC register. An RV64 microcoded processor would appear identical to the one presented in this document, except for 64-bit registers, a 64-bit bus, and some additional ALU operations to support the new instructions added to RV64.

The Enable Signals

Since the bus is shared by different components, there is a need to make sure that only one component is driving ("writing") the bus at any point in time. We do this by using tri-state buffers at the output of each component that can write to the bus. A tri-state buffer is a simple combinational logic buffer with enable control. When enable is 1, then the output of the buffer simply follows its input. When enable is 0, then the output of the buffer "floats" – i.e., regardless of its input, the tri-state buffer will not try to drive any voltage on the bus. Floating the buffer's output allows some other component to drive the bus.[2]

In this bus-based RISC-V implementation, there are four enable signals – ImmEn, ALUEn, RegEn, and MemEn – each of which are directly connected to the enable of a tri-state buffer. When setting these signals, it is important to ensure that *at most one* device is driving the bus at any time. It is possible not to assert any of the four signals, in which this case the bus will float and have an undefined value.

Special-Purpose Registers and Load Signals

In addition to the entries in the register file, this bus-based implementation contains four other special-purpose internal registers: IR (instruction register), A, B, and MA (memory address). These 32-bit registers are *positive edge-triggered* with load enable control. As shown, these registers take their data inputs from the bus. If a register's enable is asserted during a given cycle, then the value on the bus during that cycle will be copied into the register (sampled) at the *next* rising clock edge. If the enable control is 0, then register's value remains changed.

We refer to these register enable signals as *load* signals denoted by the suffix "Ld": IRLd, ALd, BLd, and MALd. In addition, the RegWr and MemWr signals are also load signals, but their exact functionality will be discussed later. It is possible to assert more than one load signal at a time, in which case the value on the bus will be loaded into all registers whose load signals are asserted.

The Instruction Register

The instruction register (IR) is used to hold the current 32-bit instruction word. The opcode and function fields (see the base instruction formats described by the RISC-V user-level ISA specification) are used by the microcode control logic to identify the instruction and dispatch to the appropriate microcode sequence. As shown in Figure H1-A, the immediate field is connected to an immediate selector and sign extender and then to the bus. Finally, as described below, the register specifier fields connect to a multiplexer that drives the register file address input.

The Immediate Selector & Sign Extender

The module labeled "Immed Select" is an immediate selector and sign extender that extracts the immediate from the IR and sign-extends the immediate to a 32-bit value. Five different immediates can be produced: ImmI, ImmU, ImmS, ImmJ, and ImmB. ImmI selects the immediate for I-type instructions; ImmU selects for the U-type format used with LUI/AUIPC instructions; ImmS selects

[2] You can also think of a tri-state buffer as an electronically controlled switch. If enable is 1, then the switch connects the input and the output as if they were connected by a wire. If enable is 0, then the input is electrically disconnected from the output. Note that the tri-state buffer is a memoryless device and is *not* the same as a latch or a flip-flop.

for the S-type format used with stores; ImmJ selects for the J-type format used with unconditional jumps; and ImmB selects for the B-type format used with conditional branches. All immediates are signed-extended.

The Arithmetic Logic Unit

The ALU takes 3 inputs: two 32-bit operand inputs, connected to the A and B registers, and an ALUOp input. ALUOp selects the operation to be performed on the operands. Assume that the ALU can perform the following operations by default, if not explicitly stated otherwise:

ALUOp	ALU Result Output
COPY_A	A
COPY_B	B
INC_A_1	A+1
DEC_A_1	A-1
INC_A_4	A+4
DEC_A_4	A-4
ADD	A+B
SUB	A-B
SLT	Signed(A) < Signed(B)
SLTU	A < B

Table H1-1: ALU Operations for Handout #5.

In order to implement the entire RISC-V ISA, we will need a few more ALU operations.

The ALU is purely combinational logic. It has two outputs, a 32-bit main result output and 1-bit zero flag output, *zero*. The result output is computed as in Table H1-1. The zero flag simply indicates if the ALU result output equals zero. If the result is 0 then *zero* is 1, otherwise *zero* is 0. For example, if A=2, B=2, and ALUOp=SUB, then the ALU result will be 0, and *zero* will be 1. This flag is used to resolve conditional branches in microcode.

The Register File

The register file contains the 32 general-purpose registers (GPRs) and the PC. The register file itself has a 6-bit address input (addr) and a single 32-bit bidirectional data port.

Two control signals determine how the register file is used during a certain cycle: RegWr and RegEn. RegWr determines whether a write operation is performed. RegEn is the enable signal for the tri-state buffer connecting the register file to the bus. If RegEn is 1, then the data from a read operation is driven onto the bus. Note that RegWr and RegEn are mutually exclusive; while it is possible for both signals to be 0 at the same time, indicating that the register file is unused that cycle, RegWr and RegEn should never be asserted simultaneously.

Figure H1-B shows how RegEn and RegWr is connected to the peripheral control circuity of the register file. The address input (addr) determines which entry in the register file is used. Read operations are assumed to be combinational – if you change the value at the addr input, then the value at the data output will change appropriately (after some delay), *even if* no clock edge occurs. Write operations, on the other hand, are positive edge-triggered, such that data from the register's data input is only stored to the selected address at the next rising clock edge.

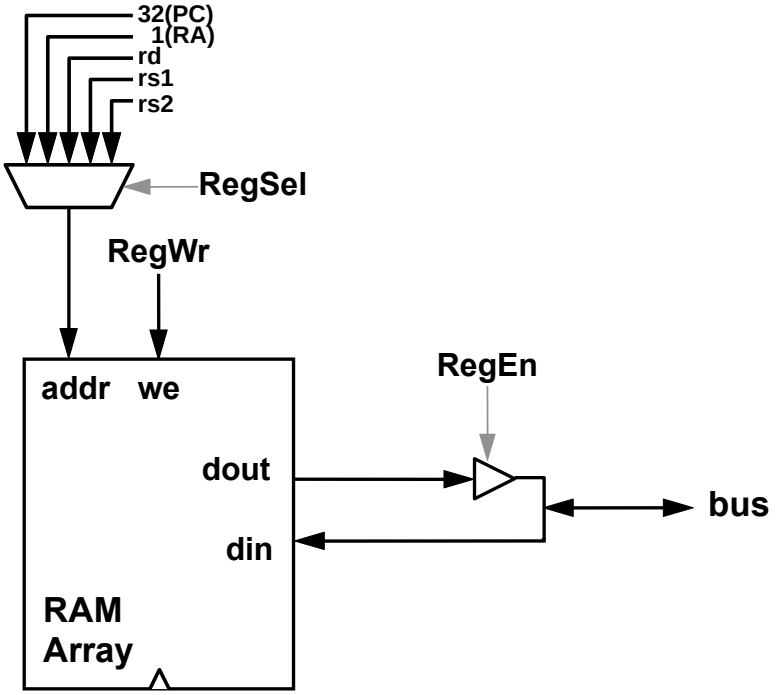


Figure H1-B: Control signals and bus connections for the register file and the memory.

While the addr input selects which register is used, the RegSel control signal selects what value is used as the addr input. As shown, RegSel controls a multiplexer that chooses between (at least) five possible values. The first two are the hardwired values 32 and 1, corresponding to the PC and the return address (RA) register, respectively. The next three (rs1, rs2, and rd) are taken from the register specifier fields of the IR. The following figure shows the corresponding locations of the rs1, rs2, and rd fields:

…	rs2	rs1	…	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

The exact meaning of these fields depends on the instruction and instruction type. Please refer to the RISC-V ISA manual and the lecture notes. Note that these fields are 5 bits wide, while addr is 6 bits wide since the register file here contains more than just the 32 GPRs. The 5-bit fields are widened to a 6-bit address by padding the MSB with a 0. When specifying the value of RegSel, you should use the symbols PC, RA, rs1, rs2, and rd.

The Memory

The memory module is very similar to the register file except that the address input is taken from an edge-triggered register, MA (memory address). Thus, it takes two steps to access a particular memory location. First, the MA is loaded with the desired memory address, and then the desired memory operation is performed on the location specified by MA. The MemWr and MemEn control signals work identically to RegWr and RegEn.

The main difference between memory module and the register file is the busy signal. Unlike the register file, the memory may take a variable number of cycles to perform a read or write (e.g., if a cache miss occurs). The busy signal indicates that the memory operation has not yet completed. The microcode can then respond to the busy signal appropriately by stalling. Note that the value of MA, as well as the data input for stores, should remain unchanged while busy is asserted.

The Clock Period and Timing Issues

We will assume that the clock period is long enough to guarantee that the results of all *combinational* logic paths are valid and stable before the setup time of any edge-triggered components attempting to latch these results.

Remember that these combinational paths include not only computational elements like the sign extender, and the ALU, but also the register file during *read* operations. Specifically, the path from addr to the data output in the register file is purely combinational. As mentioned above, if you change the value at the addr input, then the value at the data output will change appropriately (after some delay), *even if* no clock edge occurs.

Also remember that the path from data *input* to data *output* is *not* combinational. This applies not only to the register file and memory but also to other edge-triggered registers (i.e., IR, A, B, and MA). When performing a write to any of these components, the value at the data input is not sampled until the *next* rising clock edge and can thus only be read during the next clock cycle.

Finally, we will assume that there are no hold time violations.

Microprogramming on the Bus-Based RISC-V Implementation

In the past, simple CISC machines employed a bus-based architecture wherein the different components of the machine (ALU, memory, etc.) communicated through a common bus. This type of architecture is easy and inexpensive to implement but has the disadvantage of requiring all data movement between components to share the bus. Because of this bottleneck, an instruction on such an architecture typically took several cycles to execute.

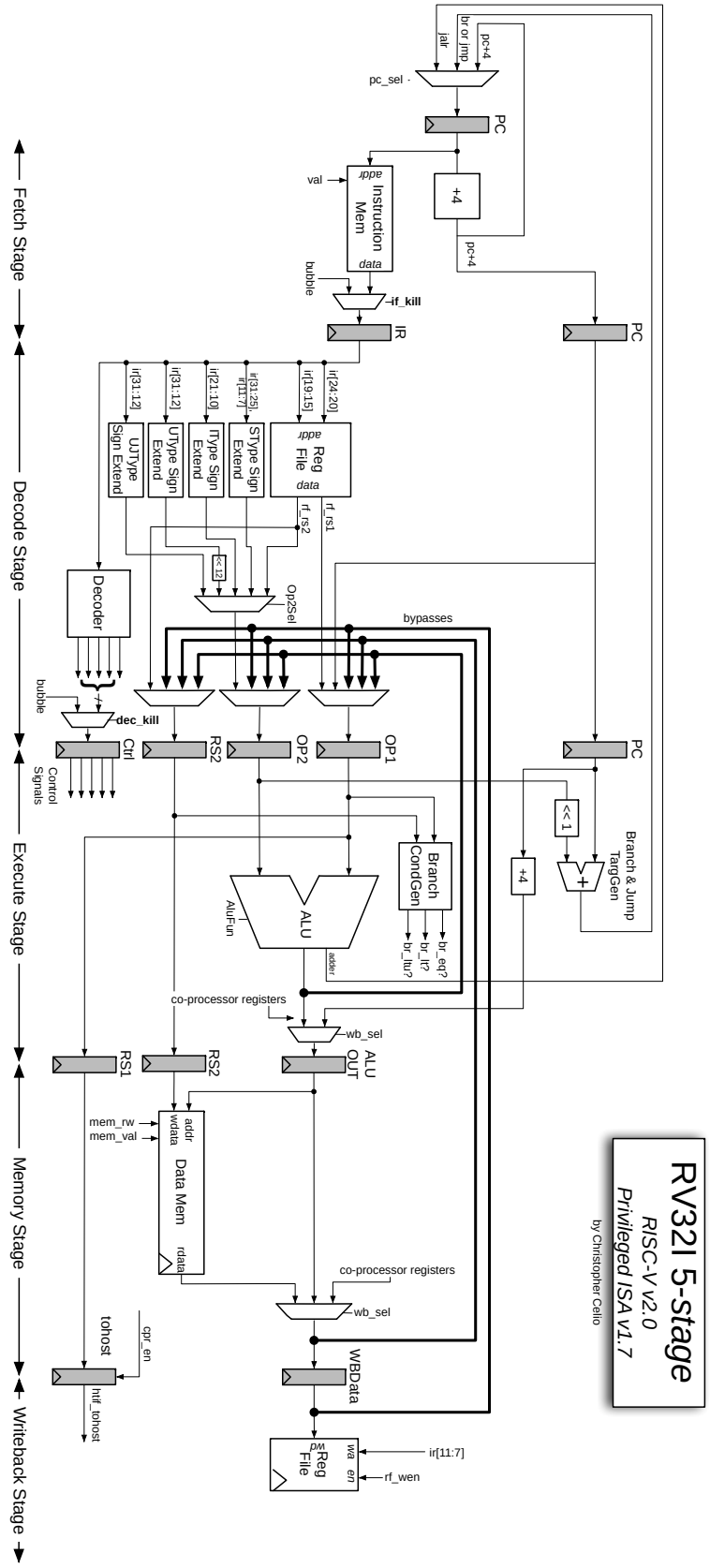
Microprogramming makes it easy to generate the control signals for such multi-cycle instructions. A microcode program is basically a finite state machine description. Each line of microcode corresponds to some state of the machine and contains a "microinstruction" which specifies the outputs for that state and the next state. The outputs are used to control the different components in the datapath. The next state depends on the current state and possibly on certain other signals (e.g., condition flags). As we shall see, this simple FSM model proves to be very powerful, allowing complex operations like conditional branches and loops to be performed.

Table H1-3 in Appendix B shows a microcode table for the bus-based RISC-V implementation. Some lines have been filled in as examples. The first column in the microcode table contains the state label of each microinstruction. For brevity, we assume that the states are listed in increasing numerical order, and we only label important states. For example, we label state FETCH0, and assume that the unlabeled line immediately following it corresponds to state (FETCH1). The second column contains a pseudo-code explanation of the microinstruction in RTL-like notation. The rest of the columns, except the last two, represent control signals that are asserted during the current cycle. 'Don't care' entries are marked with a '*'.

The last two columns specify the next state. The μBr (*micro*branch) column represents a 3-bit field with six possible values: N, J, EZ, NZ, D and S. If μBr is N (next), then the next state is simply (*current state* + 1). If it is J (jump), then the next state is *unconditionally* the state specified in the Next State column (i.e., an unconditional microbranch). If it is EZ (branch-if-equal-zero), then the next state depends on the value of the ALU's *zero* output signal (i.e., a conditional microbranch). If *zero* is asserted (== 1), then the next state is that specified in the Next State column, otherwise, it is (*current state* + 1). NZ (branch-if-not-zero) behaves exactly like EZ, except NZ branches to the next state is *zero* is not asserted (!= 1). If μBr is D (dispatch), then the FSM looks at the opcode and function fields in the IR and goes into the corresponding state. In this implementation, we assume that the dispatch goes to the state labeled (RISC-V-instruction-name + "0"). For example, if the instruction in the IR is SW, then the dispatch will go to state SW0. If μBr is S, then the next state depends on the value of the *busy?* signal. If *busy?* is asserted, then the next state is the same as the *current state* (the μPC "spins" on the same microaddress), otherwise, the next state is (*current state* + 1).

The first three lines in the table (starting at FETCH0) form the *instruction fetch* stage. These lines are responsible for fetching the next instruction opcode, incrementing the PC, and then jumping to the appropriate line of microcode based on the opcode fetched. The instruction fetch stage is performed for every instruction, and every instruction's microcode should always end by executing an instruction fetch for the next instruction. The easiest way to do this is by having the last microinstruction of an instruction's microcode include a microbranch to FETCH0. The microcode for NOP provides a simple example. For the rest of the instructions, their microcode sequences intentionally omitted; we expect students to fill out the table themselves.

RV32I 5-Stage Pipeline Diagram



Appendix A. A Cheat Sheet for the Bus-based RISC-V Implementation

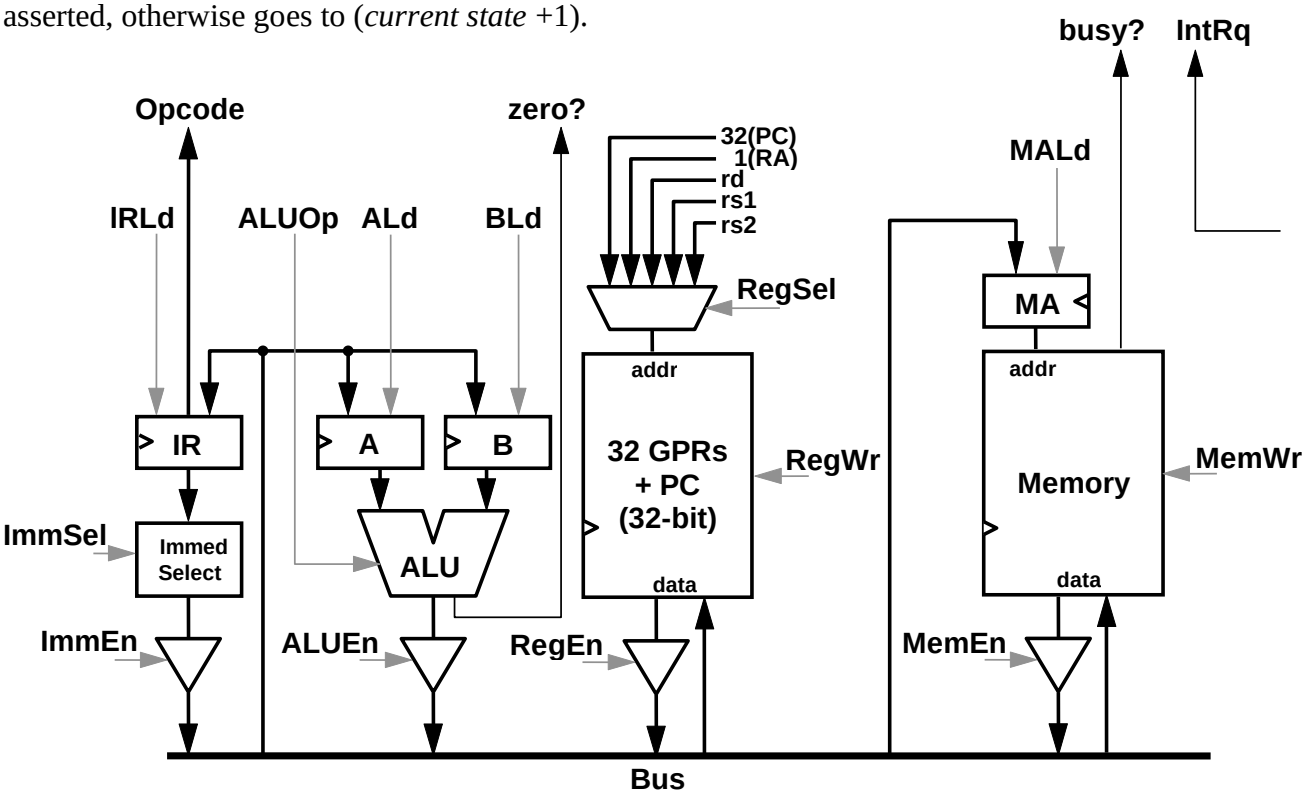
For your reference, we have reproduced the bus-based datapath diagram as well as a summary of some important information about microprogramming in the bus-based architecture.

Remember that you can use the following ALU operations:

ALUOp	ALU Result Output
COPY_A	A
COPY_B	B
INC_A_1	A+1
DEC_A_1	A-1
INC_A_4	A+4
DEC_A_4	A-4
ADD	A+B
SUB	A-B
SLT	Signed(A) < Signed(B)
SLTU	A < B

Table H1-2: Available ALU operations

Also remember that μBr (*micro*branch) column in Table H1-3 represents a 3-bit field with six possible values: N, J, EZ, NZ, D, and S. If μBr is N (next), then the next state is simply (*current state* + 1). If it is J (jump), then the next state is *unconditionally* the state specified in the Next State column (i.e., an unconditional microbranch). If it is EZ (branch-if-equal-zero), then the next state depends on the value of the ALU's *zero* output signal (i.e., a conditional microbranch). If *zero* is asserted (== 1), then the next state is that specified in the Next State column, otherwise, it is (*current state* + 1). NZ (branch-if-not-zero) behaves exactly like EZ, but instead performs a microbranch if *zero* is not asserted (!= 1). If μBr is D (dispatch), then the FSM looks at the opcode and function fields in the IR and goes into the corresponding state. If S, the μPC spins if *busy?* is asserted, otherwise goes to (*current state* +1).



Appendix B. Microcode Table for the Bus-based RISC-V Implementation

State	Pseudocode	IR Ld	Reg Sel	Reg Wr	Reg En	A Ld	B Ld	ALUOp	ALU En	MA Ld	Mem Wr	Mem En	Imm Sel	Imm En	μBr	Next State
FETCH0:	MA ← PC; A ← PC	*	PC	0	1	1	*	*	0	1	0	0	*	0	N	*
	IR ← Mem	1	*	0	0	0	*	*	0	0	0	1	*	0	S	*
	PC ← A+4	0	PC	1	0	0	*	INC_A_4	1	*	0	0	*	0	D	*
. . .																
NOP0:	microbranch back to FETCH0	*	*	0	0	*	*	*	0	*	0	0	*	0	J	FETCH0

Table H1-3 (Worksheet 1)